COMPILER GENERATION OF A LATE BINDING INTERFACE IMPLEMENTATION

COPYRIGHT NOTICE/PERMISSION

TECHNICAL FIELD

The present invention pertains to generation by a compiler of an implementation for a late binding interface. A compiler automatically generates a late binding interface implementation based upon programming language code and definition information for the late binding interface.

BACKGROUND OF THE INVENTION

To manage the complexity of long computer programs, computer programmers often adopt object-oriented programming techniques. With these techniques, a computer program is organized as multiple smaller modules called objects. These objects perform specified functions and interact with other objects in pre-defined ways. Figure 1 shows several principles of object-oriented

programming with reference to an object 100 that interacts in pre-defined ways with a client 140 (which can also be an object).

The object 100 encapsulates data 110, which represents the current state of the object 100. The data 110 can be organized as data elements such as properties of the object 100. The object 100 exposes member functions (alternatively called methods) 120, 122, and 124 that provide access to the data 110 or provide some other defined function or service. To access the data 110, the client 140 of the object 100 goes through a member function. In Figure 1, the member functions 120, 122, 124 are grouped into an interface 126. Figure 1 also includes an interface 130 (shown without member functions for simplicity).

The object 100 and the client 140 interact across the interfaces exposed by the object 100. For example, for the client 140 to invoke the member function 120, a function call in the client 140 identifies the member function 120, provides any data to be processed (e.g., input parameters for arguments), and indicates any data to be returned (e.g., output parameters for arguments, return values). Similarly, for the client 140 to set a value for a property of the data 110, the client 140 identifies the property and provides the value.

The client 140 initially identifies the member functions of the object 100 by name. Before the client 140 can access a member function of the object 100, the client's name for the member function must become associated with the actual member function in the object 100 in a binding process. Binding can occur at any of several different times, and the best time for binding depends on the situation.

When the client 140 and the object 100 are tightly integrated, it is often efficient to use early binding. With early binding, the names in the client 140 are associated with the actual respective member functions of the object 100. The client 140 directly operates upon the member functions of the object 100 through an interface.

Figure 2 shows an object 200 that provides an early binding implementation of the interface ITest. The object 200 is designed according to Microsoft Corporation's Component Object Model ["COM"] and includes an instance data structure 202, a virtual function table 210 for ITest, and method implementations 221-226. The instance data structure 202 includes a pointer 204 to the virtual function table 210 and data 208, such as properties. In early binding, a client is compiled to use the virtual function table and function signatures of ITest. A client references the early binding interface using a pointer to the pointer 204. The virtual function table 210 includes a pointer for each member function of ITest. The pointers 211-213 correspond to the standard COM IUnknown methods QueryInterface, AddReference, and Release, and point to implementations 221-223, respectively. The pointers 214-216 correspond to the other methods of ITest, and point to implementations 224-226, respectively. For additional information about COM and early binding interfaces, see Kraig Brockschmidt, Inside OLE, second edition, Microsoft Press (1995).

While early binding works well for a tightly integrated object and client, an object might need to work with a client that cannot use a custom early binding mechanism for the object. In such situations, late binding works well. With late

binding, an object exposes an intermediary interface. A client accesses late bound member functions of an object through this intermediary interface.

Late binding can be provided through numerous mechanisms. One mechanism uses tokens to identify the late bound member functions of an object. A client associates its member function names and property names with corresponding tokens. To invoke a late bound member function, the client passes the corresponding token to the intermediary interface, along with any arguments for the member function (typically packed into a generic data structure). The intermediary interface then calls the member function identified by the token. With late binding, names can be associated with tokens at run time or at compile time. If the association occurs at compile time, the tokens are still used to invoke late bound methods through the intermediary interface at run time, but a client can call the intermediary interface without first looking up a token for a name at run time.

Figure 3 shows one type of intermediary interface, a dispatch interface implemented by a COM object 300. In Figure 3, the dispatch interface provides a late binding mechanism for late bound member functions of ITest.

The object 300 includes an instance data structure 302, a virtual function table 310, and method implementations 324-327. The instance data structure includes a pointer 304 to the virtual function table 310 and data 308 (e.g., properties). The virtual function table 310 includes a pointer for each member function of ITest that is inherited from the interface IDispatch. Again, the pointers 311-313 correspond to the standard COM IUnknown methods QueryInterface,

AddReference, and Release, and point to implementations not shown in Figure 3. The pointers 314-317 correspond to the remaining IDispatch methods, and point to implementations 324-327, respectively.

By calling the method GetIDsOfNames, a client retrieves a dispatch identifier ["DISPID"] for a particular name of a late bound member function. A DISPID is a type of token that is passed to the dispatch interface to invoke a corresponding member function. In Figure 3, a client could retrieve the DISPID 18 by querying GetIDsOfNames for the DISPID of the method Score.

A client calls Invoke, passing a DISPID for a late bound method, arguments packed into a generic data structure, and other information. The Invoke implementation 327 maps DISPIDs to the implementations 224-226 for the other, non-dispatch methods of ITest. The Invoke implementation 327 calls the appropriate method for the passed DISPID. If a client passed the DISPID 18 to Invoke, the Invoke implementation 327 would call the method Score. For additional information about the dispatch interface, see Kraig Brockschmidt, Inside OLE, second edition, Chapters 14 and 15, Microsoft Press (1995).

Late binding interfaces provide flexibility and power, but not without a price. Manually programming the extra layers of code for late binding interfaces has proven to be notoriously difficult and time-consuming for programmers. Writing the corresponding client side code for packing arguments to and unpacking arguments from the generic data structures has also proven difficult.

While several attempts have been made to simplify the programming of late binding interfaces, none has been completely successful. Typically, these

attempts simplify the act of creating dispatch interface implementations, but add unnecessary run time overhead to the resulting implementations. In comparison, manually created dispatch interface implementations, though difficult to create, are relatively efficient when executing.

Microsoft Corporation provides type library support for dispatch interfaces. A type library object includes definition information about an interface, and can expose methods GetIDsOfNames and Invoke for the interface. To implement a dispatch interface in an object, a programmer can make use of these methods provided by the type library object. Although this hides the details of implementing the dispatch interface from the programmer, the resulting implementation requires numerous run time calls between the object and the type library object, which slows processing. Moreover, a type library implementation typically uses an early binding mechanism of a dual interface (an interface that has both early binding and late binding mechanisms). Essentially, the type library is used as a virtual function table, which is no more robust than a virtual function table for a custom interface -- type library implementations are not very useful for distributed computing (e.g., with objects designed according to Microsoft Corporation's Distributed Component Object Model ["DCOM"]), where the server may be several versions newer than the client, and hence type library information may be out of date. For additional information about implementing dispatch interfaces with a type library, see Shepherd et al, MFC Internals, Chapter 14, "MFC and Automation," Addison Wesley (1996).

Microsoft Foundation Classes ["MFC"] also provides some support for implementing dispatch interfaces. MFC provides a high-level software tool called a wizard that simplifies the act of creating dispatch interface implementations. The wizard manipulates the source code for an object and helps create a dispatch map, which is a table that associates names with DISPIDs. At run time, the dispatch map is queried in multiple calls between the object and a special dispatch object. Getting DISPIDs through this mechanism is slow, as is invoking late bound methods. Further, having a wizard implement the dispatch interface can lead to source code that is confusing and unstable when changed by a programmer. If the definition for a member function changes, however, the programmer may need to change the confusing and unstable wizard-generated code. Finally, while MFC provides a dispatch implementation tool for MFC, the tool does not work for other platforms. For additional information about implementing dispatch interfaces with MFC, see Shepherd et al, MFC Internals, Chapter 14, "MFC and Automation," Addison Wesley (1996).

SUMMARY OF THE INVENTION

The present invention overcomes these problems by automatically generating a late binding interface implementation based upon programming language code and definition information for the late binding interface. The definition information can be embedded in the programming language code or imported as part of a type library or other file.

Given this definition information for the late binding interface, a compiler generates potentially thousands of lines of code that the programmer previously had to write when creating a late binding interface implementation. The implementation includes code for invoking methods through a late binding mechanism, and also can include code for directly invoking the methods through an early binding mechanism.

Generating late binding interface implementations in this way is simple, systematic, and easy for the programmer to do. This technique prevents run time errors due to incomplete implementation, and a generated implementation reflects the latest definition information for the late binding interface.

Client-side code for calling a late bound method of a late binding interface can also be generated, which simplifies the packing and unpacking of generic data structures used with the late binding interface implementations.

Additional features and advantages of the invention will be made apparent from the following detailed description of an illustrative embodiment that proceeds with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a software object and a client of the software object that interact across interfaces exposed by the software object according to the prior art.

Figure 2 is a block diagram of a COM object that provides an early binding COM implementation of the interface ITest according to the prior art.

Figure 3 is a block diagram of a COM object that provides a dispatch interface implementation of the interface ITest according to the prior art.

Figure 4 is a block diagram of a computing environment that can be used to implement the illustrative embodiment.

Figure 5 is a source code listing for a file having C++ source code with embedded IDL for a dispatch interface.

Figure 6 is a source code listing for a file having C++ source code with imported definition information for a dispatch interface.

Figure 7 is a block diagram of a C++ compiler that generates dispatch interface implementations.

Figure 8 is a flow chart showing generation of a dispatch interface implementation by the compiler of Figure 7.

Figures 9a-9d are a flow chart showing generation of a dispatch interface implementation from C++ source code and definition information.

Figures 10a-10f are a source code listing representative of a compiler-generated dispatch interface implementation.

Figures 11a-11b are a flow chart showing generation of a client-side code for calling a member function of a dispatch interface.

DETAILED DESCRIPTION OF AN ILLUSTRATIVE EMBODIMENT

The illustrative embodiment of the present invention is directed to automatically generating a dispatch interface implementation. A C++ compiler automatically generates the dispatch interface implementation based upon

definition information for the dispatch interface and C++ source code for the methods invoked through the dispatch interface.

Although the illustrative embodiment addresses implementation of dispatch interfaces, the invention is also applicable to the implementation of dual interfaces, which combine dispatch interface and early binding features.

Moreover, the invention can be used with other object models (e.g., objects designed according to the Common Object Request Broker Architecture ["CORBA"] or according to Sun Microsystems' Java specifications) and other types of late binding interface implementations. In general, the present invention can be applied to software objects for which late binding interface implementations are generated from definition information and programming language code.

I. Computing Environment

Figure 4 illustrates a generalized example of a suitable computing environment 400 in which the illustrative embodiment may be implemented. Computing environment 400 is not intended to suggest any limitation as to scope of use or functionality of the invention, as the present invention may be implemented in diverse general purpose or special purpose computing environments.

With reference to Figure 4, computing environment 400 includes at least one processing unit 410 and memory 420. In Figure 4, this most basic configuration is included within dashed line 430. The processing unit 410

executes computer-executable instructions and may be a real or a virtual processor. In a multi-processing system, multiple processing units execute computer executable instructions to increase processing power. Memory 420 may be volatile memory (e.g., registers, cache, RAM), non-volatile memory (e.g., ROM, EEPROM, flash memory, etc.), or some combination of the two.

A computing environment may have additional features. For example, computing environment 400 includes storage 440, one or more input devices 450, one or more output devices 460, and one or more communication connections 470. A bus, controller, network, or other interconnection mechanism (not shown) interconnects the components of computing environment 400. Typically, operating system software (not shown) provides an operating environment for other software executing in the computing environment 400, and coordinates activities of the components of the computing environment 400.

Storage 440 may be removable or non-removable, and includes magnetic disks, magnetic tapes or cassettes, CD-ROMs, DVDs, or any other medium which can be used to store information and which can be accessed within computing environment 400.

Input device 450 may be a touch input device such as a keyboard, mouse, pen, or trackball, a voice input device, a scanning device, or another device that provides input to computing environment 400. Output device 460 may be a display, printer, speaker, or another device that provides output from computing environment 400.

Communication connection 470 enables communication over a communication medium to another computing entity. The communication medium conveys information such as computer executable instructions or other data in a modulated data signal. A modulated data signal is a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media include wired or wireless techniques implemented with an electrical, optical, RF, infrared, acoustic, or other carrier.

The invention can be described in the general context of computer readable media. Computer readable media are any available media that can be accessed within a computing environment. By way of example, and not limitation, with computing environment 400, computer readable media include memory 420, storage 440, and communication media. Combinations of any of the above also fall within the definition of computer readable media.

The invention can be described in the general context of computer executable instructions, such as those included in program modules, being executed in a computing environment on a target real or virtual processor. Generally, program modules include routines, programs, libraries, objects, classes, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The functionality of the program modules may be combined or split between program modules as desired in various embodiments. Computer-executable instructions for program modules may be executed within a local or distributed computing environment.

II. Late Binding Interfaces

The illustrative embodiment of the present invention is directed to automatically generating dispatch interface, or "dispinterface," implementations. A dispatch interface inherits from IDispatch, and includes the IUnknown methods as well as four methods: GetTypeInfoCount, GetTypeInfo, GetIDsOfNames, and Invoke. An interface that inherits from IDispatch may be a custom interface or a dual interface, in which case other, non-dispatch methods are also directly accessible through the vtable for the interface. Alternative embodiments of the present invention are directed to automatically generating implementations for other types of late binding interfaces.

The following description highlights several features of IDispatch that are relevant to the illustrative embodiment. This section indicates type and definition information needed to implement a dispatch interface. For more information about IDispatch, see Kraig Brockschmidt, Inside OLE, second edition, Microsoft Press (1995).

The methods GetIDsOfNames and Invoke provide the core functionality of a dispatch interface. The invocation of other, non-dispatch methods takes place through GetIDsOfNames and Invoke. In alternative embodiments, other types of late binding interfaces have other mechanisms for mapping late bound method names to corresponding identifiers and calling late bound methods with such corresponding identifiers through an intermediary interface.

The method GetIDsOfNames maps a member and an optional set of argument names to a corresponding set of integer dispatch identifiers ["DISPIDs"]. The DISPIDs are used in subsequent calls to Invoke. GetIDsOfNames accepts from a client an array of names and a count of the names. The first element represents the member name, while the subsequent elements represent each of the member's arguments. GetIDsOfNames returns an array, each element of which contains a DISPID corresponding to one of the names passed in. Implementation of GetIDsOfNames requires definition information about member names and corresponding DISPIDs. In alternative embodiments, instead of DISPIDs, other types of identifiers are used in a late binding interface.

The method Invoke accepts a DISPID identifying a late bound member to call. Although a client can determine this DISPID at run time with the method GetIDsOfNames, it is also possible for a client to possess this information before run time. Thus, implementation of Invoke requires definition information about members and member names, member argument names and counts, along with any associated IDL attribute information and corresponding DISPIDs.

The method Invoke also accepts an array of flag values that indicate whether the member is to be invoked as a method, retrieved as a property, changed as a property, etc. For this reason, implementation of Invoke requires definition information about these flag values for the interface.

The arguments for the operation are passed from the client to the method Invoke in a DISPPARAMS structure, which is a type of generic data structure for arguments. For a method invocation, this structure contains an array of generic

types containing the arguments and their types, a corresponding array of DISPIDs of named arguments (properties), and counts for the numbers of elements in the arrays. To implement code for unpacking arguments from a DISPPARAMS structure, a compiler requires type and definition information for member arguments. The compiler also needs this information to generate client-side code for packing arguments into a DISPPARAMS structure. Using this definition information, the method Invoke ensures that arguments passed to a late bound method are of the correct type and in the correct order. If possible, arguments passed as an incorrect type are coerced into a correct type before calling the late bound method.

The method Invoke returns a value from a late bound method in a VARIANT structure, which is a type of generic data structure for arguments. The VARIANT structure indicates a return value and a type for the return value. To implement packing and unpacking code for the VARIANT structure, the compiler again requires type and definition information for member arguments.

In alternative embodiments, late binding interfaces use other types of generic data structures to handle arguments for packing/unpacking and type coercion operations.

III. Definition Information and Source Code Files

Definition information for a dispatch interface can come from a type library or from interface definition language ["IDL"] information embedded in a C++ source code file. Given this definition information, the compiler generates

potentially thousands of lines of code that a programmer previously had to write when creating a dispatch interface implementation. Alternative embodiments of the present invention work with different types of definition information. One alternative embodiment includes a Java object described with Java definition information (e.g., a JavaBean described with an associated information bean). Another alternative embodiment includes a CORBA-compliant object described with CORBA IDL. In general, the present invention can be applied to software objects for which late binding interface implementations are generated from definition information and programming language code.

Figures 5 shows an exemplary C++ source code file 500 with embedded IDL. The file 500 includes IDL defining the dispatch interface ITest as well as C++ source code for implementing the late bound methods of ITest in an object CTest. Alternatively, different IDL tags or different programming language constructs can convey equivalent definition information.

The embedded IDL defines dispatch features and type information for the dispatch interface in a declarative fashion. Embedded IDL constructs are identified in the file 500 by brackets "[" and "]" that mark the beginning and end of a set of one or more IDL attributes, respectively. The set of IDL attributes along with brackets is termed an IDL attribute tag. In general, an IDL attribute tag can be attached to any C++ programming construct, including an entire block or program. Various forms of IDL attribute-tag syntax are allowable. An IDL attribute can simply be listed, be set to a value, be set to a list of properties, etc. Commas separate multiple attributes in a single tag.

The file 500 includes numerous embedded IDL constructs. The code portion 510 includes user-defined types enum E and struct S, each annotated with the IDL attribute "export." As such, a compiler can use type information for E and S when creating code for packing and unpacking arguments.

The code portion 520 includes IDL attributes for the dispatch interface ITest, IDL attributes for the member functions of ITest, and IDL attributes for function arguments.

The "__interface" keyword marks the interface ITest. In accordance with COM, the "__interface" keyword semantically is a struct which contains pure virtual function members (virtual function members for which no implementation is specified that must be overridden in a derived class in order to create an object). The interface ITest inherits from IDispatch, and is annotated with the "dispinterface" IDL attribute tag. Even though a dispatch interface does not directly call methods through a vtable, making the methods pure virtual forces the COM class to implement them, or else an error results at compile time. Without such a compile time check, errors might only be found at run time, which is a disadvantage of conventional dispatch interface implementation techniques.

The interface ITest includes the method Score and methods to manipulate the property Grade. The first method for Grade is a "propput" method, which sets the property Grade, while the second method for Grade is a "propget" method, which retrieves the property Grade. Each of the methods in ITest is annotated with an "id" IDL attribute to assign a DISPID. The DISPID 18 annotates the method Score. The DISPID 34 annotates the first two methods for Grade

because these methods have the same name. A flag is used to distinguish between the "propput" method and the "propget" method. These DISPIDs are used when generating implementation code for GetIDsOfNames and Invoke. Compared to type library implementations of dispatch interfaces (in which the type library essentially provides an early binding virtual function table mechanism), providing IDL information about the correspondence between DISPID and method name improves flexibility in distributed computing applications.

Arguments for the member functions of ITest are annotated with the directional IDL attributes "in," "out," and "retval." The compiler uses these IDL attributes and type information for the arguments when creating code for packing and unpacking arguments.

The code portion 530 includes C++ code for the coclass CTest that exposes the interface ITest. The code portion 530 includes a simplified implementation for each of the methods of ITest. Several IDL attributes annotate the object CTest. The IDL attribute "coclass" annotates the struct CTest as a COM class, and the other IDL attributes indicate additional information for CTest.

Figure 6 shows an exemplary listing for a C++ source code file 600 that includes a statement 620 to import a type library TestLib.tlb and a code portion 630 for the coclass CTest.

TestLib.tlb includes definition information that is roughly equivalent to the embedded IDL of the code portions 510 and 520 in Figure 5. The compiler uses this information in TestLib.tlb in the same way it uses embedded IDL information when generating a dispatch interface implementation.

IV. Programming Environment

Figure 7 shows a block diagram of a compiler environment 700 used to generate dispatch interface implementations according to the illustrative embodiment. The compiler environment 700 includes a compiler 720 that accepts C++ code 710 with embedded IDL information as input, and produces a dispatch interface implementation 790 as output. Alternatively, a compiler accepts C++ code that imports a type library with definition information. In alternative embodiments, another type of programming environment is used to create a late binding interface implementation. In one alternative embodiment, a Java compiler environment produces virtual machine instructions (e.g., bytecodes) for a late binding interface implementation.

Modules in the compiler environment 700 recognize embedded IDL constructs in C++ source code, create a unified representation of the embedded IDL and C++ source code in a parse tree, and derive semantic meaning from the embedded IDL. The compiler environment 700 also provides error detection for the C++ code 710 and embedded IDL information.

The compiler environment 700 includes a C++ compiler 720 that accepts as input a file 710 having C++ source code with embedded IDL. The C++ code includes code for implementing the late bound methods of an interface, while the IDL information defines dispatch features of the interface. The compiler environment 700 processes the file 710 in conjunction with one or more IDL attribute providers 770. Although Figure 7 depicts a single IDL attribute provider

770, the compiler 720 can work with multiple IDL attribute providers (e.g., different providers for different IDL constructs). Alternatively, the functionality of the provider 770 can be merged with the compiler 720.

A front end module 722 reads and performs lexical analysis upon the file 710. Basically, the front end module 722 reads and translates a sequence of characters in the file 710 into syntactic elements, or "tokens," indicating constants, identifiers, operator symbols, keywords, punctuation, etc.

A converter module 724 parses the tokens into an intermediate representation. For tokens from C++ source code, the converter module 724 checks syntax and groups tokens into expressions or other syntactic structures, which in turn coalesce into statement trees. Conceptually, these trees form a parse tree 732. As appropriate, the converter module 724 places entries into a symbol table 730 that lists symbol names and type information used in the file 710 along with related characteristics. A symbol table entry for a particular symbol can have a list of IDL attributes associated with it.

A state 734 tracks progress of the compiler 720 in processing the file 710 and forming the parse tree 732. For example, different state values indicate that the compiler 720 has encountered an IDL attribute, is at the start of a class definition or a function, has just declared a class member, or has completed an expression. As the compiler 720 progresses, it continually updates the state 734. The compiler 720 may partially or fully expose the state 734 to an outside entity such as the provider 770, which can then provide input to the compiler 720.

Based upon the symbol table 730 and the parse tree 732, a back end module 726 translates the intermediate representation of file 710 into output code. The back end module 726 converts the intermediate representation into instructions executable in a target processor, into memory allocations for variables, and so on. In Figure 7, the output code is executable in a real processor, but in alternative embodiments the output code is executable in a virtual processor.

The front-end module 722 and the back-end module 726 can perform additional functions, such as code optimization, and can perform the described operations as a single phase or multiples phases. Except as otherwise indicated, the modules of the compiler 720 are conventional in nature, and can be substituted with modules performing equivalent functions.

In Figure 7, the provider 770 indicates how to integrate tokens for IDL constructs into the intermediate representation, for example, adding IDL attributes to a symbol table entry for a particular symbol or manipulating the parse tree 732. Thus, embedded IDL information is associated with logically proximate programming language information in the symbol table 730 and the parse tree 732.

In Figure 7, the provider 770 is external to the compiler 720, and communicates with the compiler 720 across the interfaces 750 and 780. Figure 7 depicts a simplified interface configuration of the interface 750 exposed by the compiler 720 and the interface 780 exposed by the provider 770. Alternative interface configurations are possible.

The provider 770 includes several modules. An input module 772 receives a particular IDL attribute from a tag and parses it for parameters, values, properties, or other specifications. The interfaces 750 and 780 define how this information is passed between the compiler 720 and the provider 770.

An operation module 774 determines what must be done to implement the IDL attribute, and identifies locations where code is to be injected, or where other operations are to be carried out. "Injected code" typically includes added statements, metadata, or other elements at one or more locations, but this term also includes changing, deleting, or otherwise modifying existing source code. Injected code can be stored in the provider 770 as one or more templates 776, or in some other form. In addition, parse tree transformations may take place, such as altering the list of base classes or renaming identifiers.

An output module 778 communicates back to the compiler 720 to effect changes based upon the IDL attributes. In Figure 7, the output module 778 directly manipulates internal compiler structures such as the symbol table 730 and the parse tree 732, creating symbols, adding to the parse-tree, etc. Alternatively, the output module 778 writes injected code to an external file (not shown) or send code directly to the compiler 720 as a stream of bytes (not shown) that the compiler 720 processes. Having an IDL attribute provider instruct the compiler (e.g., at converter module 724) to perform the operations gives a measure of security -- the compiler 720 can reject or modify any request that would compromise proper functioning.

As the front end module 722 encounters IDL attribute tags in the file 710, the compiler 720 changes the state 734 appropriately and saves the IDL attribute tags in a list. This list also identifies the location of the provider 770 or any other needed attribute provider, as necessary acquiring location information from a utility such as a registry.

The compiler 720 communicates the state 734 to the provider 770. When the provider 770 detects a point at which it desires to perform an operation, it signals the compiler 720 and effects changes in one of the ways mentioned above. Thus, based upon the semantics of the embedded IDL, the provider 770 affects the states and structures of the compiler 720.

At various points during the processing of the file 710, an error checker module 740 checks for errors in the C++ source code with embedded IDL. In conjunction with the front end module 722, the error checker module 740 detects errors in lexical structure of C++ source code tokens and embedded IDL tokens. With converter module 724, error checker 740 detects any syntactical errors in the organization of C++ source code tokens and embedded IDL tokens. The error checker module 740 can also flag certain semantic errors in the embedded IDL in the C++ source code with embedded IDL.

Figure 8 shows a technique 800 for processing IDL embedded in C++ source code in a compiler environment such as that shown in Figure 7. Alternatively, technique 800 can be performed by a different configuration of modules.

After a compiler reads in (act 810) the file 805, the elements of the file 805 are processed. The compiler gets (act 820) a syntactic element of the file 805 and sets (act 822) an appropriate state for that element. The compiler determines (act 824) whether that element is for a conventional C++ construct or for an IDL construct.

If the current element is for a C++ construct, the compiler converts (act 826) that element into an intermediate language. As appropriate, the compiler handles (act 828) the element, for example, by placing a node in the parse tree or adding an entry to the symbol table. If the compiler determines (act 830) that the file 805 includes more elements, the compiler proceeds with the next element.

If the current element is for an IDL construct, the compiler gets attribute information for the IDL construct. The compiler calls (act 840) an IDL attribute provider, transmitting any parameters or other data accompanying the attribute in the construct. The IDL attribute provider parses (act 850) the passed IDL attribute information.

The IDL attribute provider executes concurrently with the compiler, and more than one IDL attribute provider can be active and executing concurrently during compilation. The IDL attribute provider is loaded upon encountering the first IDL attribute, and stays loaded for the duration of the compile operation. In this way, the IDL attribute provider acts as a domain-specific compiler plug-in that is called to parse constructs that are "registered" as part of its domain.

While executing concurrently with the compiler, the IDL attribute provider detects (act 852) the occurrence of designated events within the compiler, for

example, events relate to the state of compilation (in Figure 8, represented by a dashed arrow from act 822 to act 852). The compiler exposes a compilation state to the IDL attribute provider. Examining the state, the IDL attribute provider determines whether to do nothing or to perform (act 854) an operation. Thus, the IDL attribute provider can wait until the compiler reaches a certain state, and then perform an operation when that state is reached, for example, requesting the compiler to modify the parse tree. The IDL attribute provider then waits for another event.

The IDL attribute provider can perform different operations for different events that might occur within the compiler, and for different parameters transmitted with an IDL attribute. Among these operations are injection of statements or other program elements, possibly employing templates, and modifying or deleting code. Other operations include adding new classes, methods and variables, or modifying existing ones. Modification can include renaming and extending an object or construct. In Figure 8, dashed arrows from act 854 to acts 826 and 828 represent the passing of code, state information, instructions, or other data to the compiler as described above.

Injected code is typically located remotely from where the IDL attribute appears in the C++ source code. Code can be injected at multiple locations as well. To clarify the significance of the injected code, comments around the injected code can identify the IDL attribute for which it was injected.

The scope of an IDL attribute is not bound to the scope of its associated C++ construct (variable, class, etc., see Table 1). Rather, the scope of an IDL

attribute can extend beyond the point of its use. In most cases, however, an IDL attribute affects semantics in the context of its associated C++ construct. In Figure 5, for example, the "export" IDL attributes are coextensive with the scope of their respective type definitions, and the "dispinterface" IDL attribute operates over the ITest interface declaration.

When the file 805 has been completely processed, the compiler translates (act 870) the intermediate representation into a dispatch interface implementation 890. When the compiler finishes the compile operation, the IDL attribute provider exits (act 860).

Although Figure 8 depicts acts in a particular order, per conventional compiler techniques, many of these acts can be rearranged or performed concurrently. For example, the acts of reading the file, getting elements, determining significance, and translating to output code can be overlapped to some extent.

V. Late Binding Interface Implementation

Figures 9a-9d show a technique 900 for generating a dispatch interface implementation from C++ source code and definition information. Using a compiler to perform the technique 900, a programmer easily and systematically creates a dispatch interface implementation. Moreover, the compiler considers the latest definition information for a dispatch interface, which saves the programmer from wading through wizard-generated code to make changes.

The technique 900 can be performed by a compiler such as the one shown in Figure 7, by a different configuration of modules, or by another type of programming environment. In one alternative embodiment, the late binding interface implementation that is output by a compiler environment contains bytecode instructions for a virtual processor.

Figures 10a-10f are a source code listing 1000 representative of a compiler-generated dispatch interface implementation. Because the listing 1000 is in source code form, it is merely representative of the compiler output, which can be and typically is in a computer executable form. For the sake of presentation, the original C++ and bracketed IDL attributes are presented in boldface. The code generated based upon the semantics of the definition information for the dispatch interface is presented in lighter type.

The dispatch interface implementation in the listing 1000 lacks calls to separate dispatch interface implementations (such as those provided in a type library, another class, or a dispatch map structure). As such, the dispatch interface implementation in the listing 1000 is relatively efficient at run time, avoiding the run time overhead associated with such additional run time calls.

With reference to Figure 9a, the compiler receives (act 910) C++ code for implementing the late bound methods of the interface. The compiler also receives (act 920) definition information for dispatch features of the interface. The definition information can be embedded in the C++ code, imported as part of a type library, or otherwise provided to the compiler. The compiler parses (act 930) the C++ code and definition information, and then generates (act 940) a

dispatch interface implementation. The compiler also performs semantic checks on the dispatch interface. In alternative embodiments, other types of late binding interface implementations are created, other types of programming language are used, or other types of definition information are used.

Figure 9b shows the parsing act 930 in greater detail. When the compiler encounters a declaration for a dispatch interface, the compiler marks (act 931) the interface as a dispatch interface and records (932) dispatch attributes (e.g., DISPIDs, property flags) for the members of the interface. The compiler analyzes each member (acts 933-935), recording (act 934) dispatch attributes and types for the arguments of the member. Later, when the compiler encounters a class that implements the dispatch interface, the compiler has the necessary information to implement the methods of IDispatch for the interface.

When processing the listing of Figure 5, the compiler parses the declaration for ITest in the code portion 520. The compiler marks the underlying symbol as a dispatch interface and records the DISPIDs of each of the members of ITest. The property Grade has two member functions for the DISPID 34, but the property flag (e.g., "propput," "propget") is also recorded.

The compiler then parses each member function in the declaration of ITest, recording the attributes on function arguments. For the method Score, the compiler records the attributes for the input arguments a, b, and c. The compiler later uses type information for the arguments when generating code for unpacking and type coercing the arguments a, b, and c from a DISPPARAMS structure. For the propget method of the property Grade, the "retval" attribute indicates that the

VARIANT pVarResult argument of the method Invoke is to be filled in by the value at the argument pc. The compiler later uses this information when generating code for packing and type coercing the argument into the VARIANT structure.

When the compiler encounters the "coclass" attribute for the struct CTest in the code portion 530, the compiler gets a basic implementation for the three IUnknown methods (and a hidden implementation of IClassFactory) from an associated template library (Microsoft Corporation's Active Template Library ["ATL"]). When the compiler determines that the base class of the struct CTest is ITest, the presence of the "coclass" attribute instructs the compiler to generate an implementation for the four IDispatch methods that are not in IUnknown.

Figure 9c shows the act 940 of generating a dispatch interface implementation in greater detail. The compiler generates (act 942) non-IDispatch code such as that represented in the code portion 1010. The code portion 1010 includes code provided through ATL for several base classes of CTest. The code portion 1010 also includes code for the late bound methods of the interface ITest (put_Grade, get_Grade, and Score), which comes from the original C++ code.

Next, the compiler generates (act 950) code to implement the method Invoke, such as that represented as the code portion 1020 in Figures 10b and 10c. With reference to Figure 9d, the compiler generates (act 951) boilerplate code, such as the code portions 1022 and a switch statement. Although the switch statement switches on DISPID, for DISPID 34 "wFlags" is checked to distinguish between put_Grade and get_Grade.

For each late bound member of the interface (acts 952-956), the compiler generates (act 953) unpacking and type coercing code for any input arguments. The code portion 1024 is for placing values from the DISPPARAMS structure into i1, i2, and i3. In this code portion, macros of the form V_<some type>(native data) represent the unpacking and type coercing. Microsoft Corporation's OLE provides a set of macros for converting between types for VARIANTs in this way. The native data from the DISPPARAM structure, as packed by the client, is coerced as necessary into a type appropriate for passing to the late bound method. The code portions 1034 and 1044 include code for unpacking and type coercing arguments for put_Grade and get_Grade, respectively.

The compiler next generates (act 954) code for calling the late bound method corresponding to the DISPID and property flags passed to the method Invoke. This code uses the standard signature for invoking the method. In the code portion 1026, the method Score is called with the arguments i1, i2, and i3, and an HRESULT hr is returned. The code portions 1036 and 1046 include code for calling put_Grade and get_Grade, respectively.

The compiler then generates (act 955) packing and type coercion code for any return values. The code portion 1028 is for placing an error value and the value hr into the VARIANT at pVarResult. In this code portion, the packing and type coercing are again represented as a macro. The code portion 1048 includes code for packing and type coercing return values for get_Grade. The method put_Grade, as a property put, does not have a return value.

After generating code for the method Invoke, the compiler generates (act 960) code for the method GetIDsOfNames. Generally, the compiler establishes a correspondence between late bound members of the dispatch interface and corresponding DISPIDs. The code portion 1050 includes code for initializing an array of names "names[]" to "Grade" and "Score." This code portion also includes code for initializing a corresponding array of DISPIDs to 34 and 18. Each name in a passed in array of names is compared to each name in the initialized array of names, and a DISPID is assigned when a match occurs.

Next, the compiler generates (act 970) code for the method GetTypeInfo and generates (act 980) code for the method GetTypeInfoCount. For this purpose, the compiler generates code that locates a type library and an ITypeInfo interface corresponding to the dispatch interface. If a type library does not exist, stub implementations are generated for GetTypeInfoCount and GetTypeInfo. The code portions 1060 and 1070 show the implementations for GetTypeInfoCount and GetTypeInfo, respectively. Each of these calls TypeInfoHelper, a method implemented in CTest at the code portion 1080. The method TypeInfoHelper accepts an identifier for the dispatch interface, locates and loads a type library, gets the ITypeInfo interface for the dispatch interface, and returns the ITypeInfo interface.

Although Figures 9a-9d depict acts in a particular order, per conventional implementation techniques, many of these acts can be rearranged. For example, the acts of generating code for particular dispatch methods can be rearranged.

VI. Client Side Call Site Code

Figures 11a and 11b show a technique 1100 for generating client-side code for calling a late bound method of a late binding interface. The technique 1100 can be performed by a compiler such as the one shown in Figure 7, or can be performed by a different configuration of modules. For alternative embodiments, client-side code appropriate for a particular type of late binding interface is generated.

The compiler receives (act 1110) C++ code for calling a method of an interface. The compiler also receives (act 1120) definition information for dispatch features of the interface. The compiler parses (act 1130) the C++ code and definition information, as described above with respect to Figure 9b.

The compiler determines (act 1140) if the method is part of the interface per se (i.e., in the vtable for the interface), and if it is, the compiler generates (act 1150) code for calling the method through an early binding mechanism.

If the method is not part of the interface per se, the compiler checks (act 1160) whether the interface is a dispatch interface. If so, the compiler generates (act 1170) code for calling the late bound method. To do this, the compiler first generates (act 1171) code for calling GetIDsOfNames. If the compiler already has DISPID information, it can skip this step. The compiler then determines (act 1172) whether the call should be to a method, a property put, a property get, etc. by checking the corresponding indicators in the definition information.

Next, the compiler generates (act 1173) code for packing any arguments for the late bound method into a DISPPARAMS structure. For this purpose, the

compiler uses type information and dispatch interface information for the function arguments presented in the definition of the dispatch interface.

The compiler generates (act 1174) code for calling the late bound method through the method Invoke, passing the corresponding DISPID, the DISPPARAMS structure, a pointer to the VARIANT structure for the return value, and other information.

For a method call or a property get, the compiler then generates (act 1175) code for unpacking any returned arguments for the late bound method from a VARIANT structure returned from Invoke. For this purpose, the compiler again uses information presented in the definition of the dispatch interface.

Although Figures 11a and 11b depict acts in a particular order, per conventional code generation techniques, many of these acts can be rearranged.

Having described and illustrated the principles of our invention with reference to an illustrative embodiment, it will be recognized that the illustrative embodiment can be modified in arrangement and detail without departing from such principles. It should be understood that the programs, processes, or methods described herein are not related or limited to any particular type of computing environment, unless indicated otherwise. Various types of general purpose or specialized computing environments may be used with or perform operations in accordance with the teachings described herein. Elements of the illustrative embodiment shown in software may be implemented in hardware and vice versa.

In view of the many possible embodiments to which the principles of our invention may be applied, we claim as our invention all such embodiments as may come within the scope and spirit of the following claims and equivalents thereto.